Contents

1 Leo-Babel Parameters **1**

2 Why there is no echo command lines to stdout **1**

3 Why isn't Python 2 supported? **2**

4 How To Live-Stream Stdout and Stderr **2**
4.1 Achieving Non-Hanging Reads 2
4.2 Unbuffering Stdout and Sterr 2
4.3 Suggestions that Don't Work 3

5 Log Pane Tab for Real Time Streaming Output **3**

6 Supported OS's **3**

7 g.IdleTime() **3**

8 Automated Testing **4**
8.1 |Tests| nodes 4
8.2 tests/tests.py 4
8.3 tests.leo 4
8.4 Evaluating test results 4

Note: This node's body text is ignored when writing this file.

The @others directive is not required.

1 Leo-Babel Parameters

Beware: It is somewhat confusing that there is one parameter name used to set a parameter using the standard Leo-Editor parameter setting code and another parameter name for setting the same parameter in a Babel Script.

Example: Settings node: @color Leo-Babel-stdout = @color Leo-Babel-stdout = #996633 Babel script: leo_babel_stdout = @color Leo-Babel-stdout = #996633

This is because the Leo-Editor standard for naming parameters requires the capitalization and the hyphens, but a Python variable name can't contain hyphens and the usual convention is for a variable name to begin with a lower-case letter.

2 Why there is no echo command lines to stdout

Leo-Babel writes all the lines of the script to disk file, then Leo-Babel creates a subprocess in which the appropriate interpreter (Python or Bash) executes the entire script. Hence, Leo-Babel has no way to echo each line of the script just before it is executed.

3 Why isn't Python 2 supported?

The function NamedTemporaryFile() in library module tempfile has keyword argument "buffering" only for Python 3, not Python 2. Leo-Babel uses the buffering parameter. Hence, Leo-Babel refuses to initialize when Python 2 interprets the Leo-Editor code.

The Python 2 ImportError exception does not report the name of the module causing the exception. Hence, if Python 2 interpreted the Leo-Editor and Leo-Babel code, then Leo-Babel could not report the failed dependency. But this would only be a minor incovenience. It would not prevent any major Leo-Babel service.

4 How To Live-Stream Stdout and Stderr

4.1 Achieving Non-Hanging Reads

By default, reading stdout or stderr in Python hangs. In some cases it hangs till there is data to be read. In other cases it hangs till the subprocess terminates. Leo-Babel's scheme for achieving non-hanging reads of stdout and stderr while the script is still executing is taken from:

http://stackoverflow.com/questions/18421757/live-output-from-subprocess-command

Essentially the scheme is to redirect both stdout and stderr to binary disk files. I was very surprised that such a simple scheme works. At least it works for Ubuntu 16.04. There are four or five other schemes suggested at various places on the Internet that do NOT work for Ubuntu 16.04.

Several people also suggest using asyncio. This may well work, but the asyncio scheme is orders of magnitude more complex than using binary disk files; and asyncio is only available for Python 3; and asyncio is still being revised.

4.2 Unbuffering Stdout and Sterr

The Bash interpreter never buffers stdout or stderr, so the Bash interpreter works with Leo-Babel by default.

On the other hand, the Python interpreter by default buffers stdout. The Python interpreter command line option -u turns off buffering for stdout and/or stderr if they are binary streams. So Leo-Babel uses the -u option and consequently it can live-stream Python scripts that do NOT flush() after writing to stdout or stderr.

But if you invoke Bash to execute a script that is a Python script, then Python buffers stdout, so the fact that Bash never buffers stdout doesn't help you. In this case, you need to put explicit flushes in your Python script in order to live stream stdout.

4.3 Suggestions that Don't Work

There is a Linux utility named unbuffer. When this utility is applied to the Python interpreter like "unbuffer python script", then all writes by "script" to stdout or stderr are unbuffered — but stderr is redirected to stdout. I consider this a bug.

There is a Linux utility named stdbuf. Some people claim that "stdbuf -i0 -o0 -e0 command" makes stdin, stdout, and stderr all unbuffered for "command". So far as I can tell, stdbuf has no effect on the Python interpreter.

5 Log Pane Tab for Real Time Streaming Output

Since there may a great deal of script output, it may be desirable to stream it to a log pane tab dedicated to this purpose instead of to the general purpose Log tab. On the other hand, since Leo-Editor error messages are printed to the Log tab, I prefer to have the Log tab visible as much as possible. On the other hand, if a Leo-Editor error message switched to the Log tab and output to the Babel tab did NOT switch back to the Babel tab, then I would see nothing wrong in a separate Babel tab.

6 Supported OS's

Currently Leo-Babel only works on Linux systems. Windows adding Window suport would be difficult. Some facilities may be impossible to implement under Windows.

Since I never use Windows, I will not be the person making Leo-Babel work under Windows.

7 g.IdleTime()

Leo-Editor does NOT implement g.idleTime() for LeoBridge. So for testing purposes, Leo-Babel implements g.idleTime() as part of the Leo-Babel test code.

The interface that Leo-Babel implements is:

itObj = g.IdleTime(handler, delay=nnn, tag=None)

itObj.start() itObj.stop()

handler(itObj) – The itObj is always the first argument passed to the handler.

8 Automated Testing

8.1 |Tests| nodes

Each test that is run automaticly has a Babel root node that is a child of a node whose headline ends with " |Tests|" (quote marks not included).

8.2 tests/tests.py

Command line:

tests.py <example.leo> <results.txt>

- <example.leo> Pathname of a Leo-Editor file containing tests to run.
- <results.txt> Pathname of a flat text file to contain the test results.
  - If this file doesn't exist, it is created.
  - If this file exists, its contents are overwritten.

8.3 tests.leo

Select the node with headline "One example file, one results file", then hit Ctrl-B.

You are then prompted to select a Leo-Editor file containing tests and a flat text file to contain the test results.

Then all the tests in the selected Leo-Editor file are executed.

8.4 Evaluating test results

Use meld or xxdiff to compare various test runs.

For example, the results of running the same tests using tests.py and using the GUI should be essentially the same.

You should keep a copy of each test results that you are convinced are good. Then new test results that are essentially the same as a good test results are also good.